Solver-Aided Programming I

Emina Torlak
emina@cs.washington.edu
Topics

What is this course about?

Course logistics

Getting started with solver-aided programming!
Tools for building better software, more easily
Tools for building **better software**, more easily

*more reliable, efficient, secure*
Tools for building **better software**, more easily

automated verification, synthesis, debugging, based on satisfiability solvers
Tools for building better software, more easily

automated verification, synthesis, debugging, based on satisfiability solvers

“solver-aided tools”
By the end of this course, you’ll be able to build solver-aided tools for any domain!
By the end of this course, you’ll be able to build solver-aided tools for any domain!
logistics

Topics, structure, people
Course overview

program question

logic

automated reasoning engine
Course overview

program question

verifier, synthesizer, fault localizer

logic

SAT, SMT, model finders
Course overview

program question

verifier, synthesizer, fault localizer

logic

SAT, SMT, model finders

Drawing from “Decision Procedures” by Kroening & Strichman
Course overview

program question

verifier, synthesizer, fault localizer

logic

SAT, SMT, model finders

Drawing from “Decision Procedures” by Kroening & Strichman
Course overview

**program**  **question**

**verifier,**  **synthesizer,**  **fault localizer**

**logic**

**SAT, SMT, model finders**

---

*Drawing from “Decision Procedures” by Kroening & Strichman*
Grading

3 individual homework assignments (75%)
- conceptual problems & proofs (TeX)
- implementations (Racket)
- completed on your own (may discuss HWs with course staff only)

Course project (25%)
- build a computer-aided reasoning tool for a domain of your choice
- teams of 2-3 people
- see the course web page for timeline, deliverables and other details
Reading and references

Recommended readings posted on the course web page

• Complete each reading before the lecture for which it is assigned
• If multiple papers are listed, only the first is required reading

Recommended text books

• Bradley & Manna, *The Calculus of Computation*
• Kroening & Strichman, *Decision Procedures*
Advice for doing well in 507

Come to class (prepared)

• Lecture slides are enough to teach from, but not enough to learn from

Participate

• Ask and answer questions

Meet deadlines

• Turn homework in on time
• Start homework and project sooner than you think you need to
• Follow instructions for submitting code (we have to be able to run it)
• No proof should be longer than a page (most are ~1 paragraph)
People

Emina Torlak
PLSE
CSE 596

instructor

Sorawee Porncharoenwase
PLSE
CSE 486
OH Th 2-3pm

TA
People

Emina Torlak  
PLSE  
CSE 596  
By appointment

Sorawee Porncharoenwase  
PLSE  
CSE 486  
OH Th 2-3pm

instructor
TA
students!

Your name  
Research area
Solver-aided programming in two parts: (1) getting started and (2) going pro
A programming model that integrates solvers into the language, providing constructs for program verification, synthesis, and more.

**Solver-aided programming in two parts:**
(1) getting started and (2) going pro
A programming model that integrates solvers into the language, providing constructs for program verification, synthesis, and more.

Solver-aided programming in two parts: (1) getting started and (2) going pro

How to use a solver-aided language: the workflow, constructs, and gotchas.
A programming model that integrates solvers into the language, providing constructs for program verification, synthesis, and more.

**Solver-aided programming in two parts:**
(1) **getting started** and (2) **going pro**

How to use a solver-aided language: the workflow, constructs, and gotchas.

How to build your own solver-aided tool via direct symbolic evaluation or language embedding.
A programming model that integrates solvers into the language, providing constructs for program verification, synthesis, and more.

**Solver-aided programming in two parts:**
(1) **getting started** and (2) **going pro**

How to use a solver-aided language: the workflow, constructs, and gotchas.

How to build your own solver-aided tool via direct symbolic evaluation or language embedding.
Solver-aided programming in two parts: (1) **getting started** and (2) going pro.

How to use a solver-aided language: the **workflow**, constructs and gotchas.

How to build your own solver-aided tool via direct symbolic evaluation or language embedding.

A programming model that integrates solvers into the language, providing constructs for program verification, synthesis, and more.
Classic programming: from spec to code
Classic programming: check code against spec

check the specification on concrete inputs

P(x) {
  ...
  ...
} assert safe(2, P(2))
Solver-aided programming: add *symbolic* values

The symbolic value \( x \) stands for an arbitrary integer.

```
P(x) {
    ...
    ...
}
assert safe(x, P(x))
```

**check the specification on symbolic inputs**
Solver-aided programming: query code against spec

The symbolic value \( x \) stands for an arbitrary integer.

The runtime uses the solver to determine the concrete meaning of \( x \) in response to solver-aided queries.
Solver-aided programming: query code against spec

\[ P(x) \{
\text{...}
\text{...}
\}
\text{assert safe}(x, P(x)) \]
Find an input on which the program fails.

P(x) {
  ...
  ...
}  
assert safe(x, P(x))

∃x . ¬safe(x, P(x))

Solver-aided programming: verify code against spec

verify debug solve synthesize

42

solver-aided language

SMT solver

∃x . ¬safe(x, P(x))
Solver-aided programming: *debug* code against spec

Find an input on which the program fails. Localize bad parts of the program.

```java
P(x) { 
    v = x + 2
    ...
    assert safe(x, P(x))
}
```

∃x . ¬safe(x, P(x))

x = 42 ∧ safe(x, P(x))

*SMT solver*
Find an input on which the program fails.
Localize bad parts of the program.
Find values that repair the failing run.

verify  
debug  
solve  
synthesize

\[
P(x) \{ 
  v = \text{choice()} 
  \ldots 
\} 
\text{assert safe}(x, P(x))
\]

\[
\exists x . \neg \text{safe}(x, P(x)) \\
x = 42 \land \text{safe}(x, P(x)) \\
\exists v . \text{safe}(42, P_v(42))
\]
Solver-aided programming: *synthesize* code from spec

Find an input on which the program fails.
Localize bad parts of the program.
Find values that repair the failing run.
Find code that repairs the program.

```
P(x) {
    v = ??
    ...
}
assert safe(x, P(x))
```

```
x-2
```

SMT solver

solver-aided language

\[ \exists x . \neg \text{safe}(x, P(x)) \]

\[ x = 42 \land \text{safe}(x, P(x)) \]

\[ \exists v . \text{safe}(42, P_v(42)) \]

\[ \exists e . \forall x . \text{safe}(x, P_e(x)) \]
Use **assertions** and **symbolic values** to express the specification.

Ask **queries** about program behavior (on arbitrary inputs) with respect to the specification.
A programming model that integrates solvers into the language, providing constructs for program verification, synthesis, and more.

**Solver-aided programming in two parts:**

(1) **getting started** and (2) **going pro**

How to use a solver-aided language: the workflow, **constructs**, and gotchas.

How to build your own solver-aided tool via direct symbolic evaluation or language embedding.
Rosette extends Racket with solver-aided constructs

= +

(define-symbolic id type)
(define-symbolic* id type)
(assert expr)
(verify expr)
(debug [type ...+] expr)
(solve expr)
(synthesize
 #:forall expr
 #:guarantee expr)

symbolic values
assertions
queries
Rosette extends **Racket** with solver-aided constructs

“A programming language for creating new programming languages”

A modern descendent of Scheme and Lisp with powerful macro-based meta programming.

- (define-symbolic id type)
- (define-symbolic* id type)
- (assert expr)
- (verify expr)
- (debug [type ...+] expr)
- (solve expr)
- (synthesize #:forall expr #:guarantee expr)
Rosette extends **Racket** with solver-aided constructs

```scheme
#lang rosette
(define-symbolic id type)
(define-symbolic* id type)
(assert expr)
(verify expr)
(debug [type ...+] expr)
(solve expr)
(synthesize
  #:forall expr
  #:guarantee expr)
```

symbolic values

assertions

queries
**Rosette constructs: define-symbolic**

define-symbolic creates a fresh symbolic constant of the given type and binds it to the variable id.

```plaintext
(define-symbolic id type)
(define-symbolic* id type)
(assert expr)
(verify expr)
(debug [type ...+] expr)
(solve expr)
(synthesize #:forall expr #:guarantee expr)

> (define-symbolic x integer?)
```
Rosette constructs: define-symbolic

A type that is efficiently supported by SMT solvers: booleans, integers, reals, bitvectors, uninterpreted functions.

define-symbolic creates a fresh symbolic constant of the given type and binds it to the variable id.

> (define-symbolic x integer?)

(define-symbolic id type)
(define-symbolic* id type)
(assert expr)
(verify expr)
(debug [type ...+] expr)
(solve expr)
(synthesize #:forall expr #:guarantee expr)
Rosette constructs: define-symbolic

A type that is efficiently supported by SMT solvers: booleans, integers, reals, bitvectors, uninterpreted functions.

**define-symbolic** creates a fresh symbolic constant of the given type and binds it to the variable **id**.

- `(define-symbolic id type)`
- `(define-symbolic* id type)`
- `(assert expr)`
- `(verify expr)`
- `(debug [type ...] expr)`
- `(solve expr)`
- `(synthesize #:forall expr #:guarantee expr)`

Symbolic values of a given type can be used just like concrete values of that type.
Rosette constructs: define-symbolic

A type that is efficiently supported by SMT solvers: booleans, integers, reals, bitvectors, uninterpreted functions.

(define-symbolic id type)
(define-symbolic* id type)
(assert expr)
(verify expr)
(debug [type ...+] expr)
(solve expr)
(synthesize #:forall expr #:guarantee expr)

**define-symbolic** creates a fresh symbolic constant of the given type and binds it to the variable *id*.

> (define (same-x)
>     (define-symbolic x integer?)
>     x)
> (same-x)
> (same-x) id is bound to the same constant every time define-symbolic is evaluated.

Symbolic values of a given type can be used just like concrete values of that type.
Rosette constructs: define-symbolic*

A type that is efficiently supported by SMT solvers: booleans, integers, reals, bitvectors, uninterpreted functions.

(define-symbolic id type)
(define-symbolic* id type)
(assert expr)
(verify expr)
(debug [type ...+] expr)
(solve expr)
(synthesize #:forall expr #:guarantee expr)

define-symbolic* creates a fresh symbolic constant of the given type and binds it to the variable id.

> (define (new-x)
>     (define-symbolic* x integer?)
>     x)
> (new-x)
x$0
> (new-x)
x$1
> (eq? (new-x) (new-x))
(= x$2 x$3)

id is bound to a different constant every time define-symbolic* is evaluated.

Symbolic values of a given type can be used just like concrete values of that type.
Rosette constructs: creating complex symbolic values

(define-symbolic id type)
(define-symbolic* id type)
(assert expr)
(verify expr)
(debug [type ...+] expr)
(solve expr)
(synthesize #:forall expr #:guarantee expr)

define-symbolic(*) can be used to create bounded symbolic instances of complex data types.
Rosette constructs: creating complex symbolic values

(define-symbolic* id type)
(define-symbolic* id type)
(assert expr)
(verify expr)
(debug [type ...+] expr)
(solve expr)
(synthesize #:forall expr #:guarantee expr)

(define-symbolic(*) can be used to create bounded symbolic instances of complex data types.

> (define-symbolic* xs integer? [4])
> xs
(list xs$0 xs$1 xs$2 xs$3)

A concrete list of 4 symbolic integers; this is just a short-hand for evaluating define-symbolic* 4 times and collecting the results into a list.
Rosette constructs: creating complex symbolic values

\[
\text{define-symbolic}(\text{id, type})\]

\[
\text{(define-symbolic}* id type)\]

(assert expr)

(verify expr)

(debug [type ...+] expr)

(solve expr)

(synthesize #:forall expr #:guarantee expr)

\[
> \text{(define-symbolic}* xs integer? [4])
\]

> xs

(list xs$0 xs$1 xs$2 xs$3)

> (define-symbolic* len integer?)

> (take xs len)

\{
[= 0 len$0) ()

[= 1 len$0) (xs$0)

[= 2 len$0) (xs$0 xs$1)

[= 3 len$0) (xs$0 xs$1 xs$2)

[= 4 len$0) (xs$0 xs$1 xs$2 xs$3)]\}

define-symbolic(*) can be used to create *bounded* symbolic instances of complex data types.

A symbolic list of length up to 4, consisting of symbolic integers.
**Rosette constructs: assert**

assert checks that expr evaluates to a true value.

> (assert (>= 2 1)) ; passes
> (assert (< 2 1)) ; fails

assert: failed
Rosette constructs: assert

\[
\begin{align*}
&\text{(define-symbolic id type)} \\
&\text{(define-symbolic* id type)} \\
&(\text{assert expr)} \\
&(\text{verify expr)} \\
&(\text{debug [type ...+] expr)} \\
&(\text{solve expr)} \\
&(\text{synthesize #:forall expr #:guarantee expr)}
\end{align*}
\]

**assert** checks that **expr** evaluates to a true value.

```scheme
> (assert (>= 2 1)) ; passes  
> (assert (< 2 1)) ; fails  
assert: failed
```

```scheme
> (define-symbolic* x integer?) 
> (assert (>= x 1))
```

Symbolic **expr** gets added to the assertion store. Its meaning (true or false) is eventually determined by the solver in response to queries.
Rosette constructs: assert

(assert expr)

(assert (>= 2 1)) ; passes
(assert (< 2 1)) ; fails

assert checks that expr evaluates to a true value.

(define-symbolic id type)
(define-symbolic* id type)
(assert expr)

(define-symbolic* x integer?)
(assert (>= x 1))
(asserts)
(list (<= 1 x$0) ...)

Symbolic expr gets added to the assertion store. Its meaning (true or false) is eventually determined by the solver in response to queries.
Rosette constructs: from assert to verify

Do poly and fact produce the same output on all inputs?

```
(define (poly x)
 (+ (* x x x x) (* 6 x x x)
    (* 11 x x) (* 6 x)))

(define (fact x)
 (* x (+ x 1) (+ x 2) (+ x 2)))

(define (same p f x)
 (assert (= (p x) (f x))))

; some tests ...
> (same poly fact 0) ; pass
> (same poly fact -1) ; pass
> (same poly fact -2) ; pass
```


**Rosette constructs: verify**

*verify* searches for a binding of symbolic constants to concrete values that causes at least one assertion in *expr* to fail.

```scheme
(define-symbolic id type)
(define-symbolic* id type)
(assert expr)
(verify expr)
(debug [type ...+] expr)
(solve expr)
(synthesize
 #:forall expr
 #:guarantee expr)
```

Do poly and fact produce the same output on all inputs?

```scheme
(define (poly x)
 (+ (* x x x x) (* 6 x x x)
 (* 11 x x) (* 6 x)))

(define (fact x)
 (* x (+ x 1) (+ x 2) (+ x 2)))

(define (same p f x)
 (assert (= (p x) (f x))))

; some tests ...
> (same poly fact 0) ; pass
> (same poly fact -1) ; pass
> (same poly fact -2) ; pass
```
**Rosette constructs: verify**

`verify` searches for a binding of symbolic constants to concrete values that causes at least one assertion in `expr` to fail.

```
(define-symbolic id type)
(define-symbolic* id type)
(assert expr)
(verify expr)
(debug [type ...+] expr)
(solve expr)
(synthesize
  #:forall expr
  #:guarantee expr)
```

Do poly and fact produce the same output on all inputs?

```
(define (poly x)
  (+ (* x x x x) (* 6 x x x) (* 11 x x) (* 6 x)))

(define (fact x)
  (* x (+ x 1) (+ x 2) (+ x 2)))

(define (same p f x)
  (assert (= (p x) (f x))))

> (define-symbolic i integer?)
> (verify (same poly fact i)))
```
Rosette constructs: verify

**verify** searches for a binding of symbolic constants to concrete values that causes at least one assertion in **expr** to fail.

```scheme
(define-symbolic id type)
(define-symbolic* id type)
(assert expr)
(verify expr)
(debug [type ...+] expr)
(solve expr)
(synthesize #:forall expr #:guarantee expr)
```

Do poly and fact produce the same output on all inputs?

```scheme
(define (poly x)
  (+ (* x x x x) (* 6 x x x)
      (* 11 x x) (* 6 x)))

(define (fact x)
  (* x (+ x 1) (+ x 2) (+ x 2)))

(define (same p f x)
  (assert (= (p x) (f x))))

> (define-symbolic i integer?)
> (verify (same poly fact i)))
(model [i -6])
```

No! The solver finds a concrete **counterexample** to the assertion in **same**.
**Rosette constructs: verify**

**verify** searches for a binding of symbolic constants to concrete values that causes at least one assertion in expr to fail.

```
(define-symbolic id type)
(define-symbolic* id type)
(assert expr)
(verify expr)
(debug [type ...+] expr)
(solve expr)
(synthesize #:forall expr #:guarantee expr)
```

Do poly and fact produce the same output on all inputs?

```
(define (poly x)
  (+ (* x x x x) (* 6 x x x)
      (* 11 x x) (* 6 x)))

(define (fact x)
  (* x (+ x 1) (+ x 2) (+ x 2)))

(define (same p f x)
  (assert (= (p x) (f x))))

> (define-symbolic i integer?)
> (define cex
    (verify (same poly fact i)))
> (evaluate i cex)
-6
```

We can store bindings in variables and evaluate arbitrary expressions against them.
**Rosette constructs: verify**

*verify* searches for a binding of symbolic constants to concrete values that causes at least one assertion in *expr* to fail.

```
(define-symbolic id type)
(define-symbolic* id type)
(assert expr)
(verify expr)
(debug [type ...+] expr)
(solve expr)
(synthesize #:forall expr #:guarantee expr)
```

The assertions encountered while evaluating *expr* are removed from the asserts store once a query (such as *verify*) completes.

---

Do poly and fact produce the same output on all inputs?

```
(define (poly x)
  (+ (* x x x x) (* 6 x x x)
      (* 11 x x) (* 6 x)))

(define (fact x)
  (* x (+ x 1) (+ x 2) (+ x 2)))

(define (same p f x)
  (assert (= (p x) (f x))))

> (define-symbolic i integer?)
> (define cex
    (verify (same poly fact i)))
> (asserts)
(list)
```
**Rosette constructs: from verify to debug**

Why do `poly` and `fact` output different values on the input -6?

```
(define (poly x)
  (+ (* x x x x) (* 6 x x x)
      (* 11 x x) (* 6 x)))

(define (fact x)
  (* x (+ x 1) (+ x 2) (+ x 2)))

(define (same p f x)
  (assert (= (p x) (f x))))
```
Rosette constructs: from verify to debug

**debug** searches for a minimal set of expressions of the given types that cause the evaluation of **expr** to fail.

```
(define-symbolic id type)
(define-symbolic* id type)
(assert expr)
(verify expr)
(debug [type ...+] expr)
(solve expr)
(synthesize
  #:forall expr
  #:guarantee expr)
```

Why do **poly** and **fact** output different values on the input -6?

```
(define (poly x)
  (+ (* x x x x) (* 6 x x x) (* 11 x x) (* 6 x)))

(define (fact x)
  (* x (+ x 1) (+ x 2) (+ x 2)))

(define (same p f x)
  (assert (= (p x) (f x))))
```
**Rosette constructs: debug**

- `debug` searches for a minimal set of expressions of the given types that cause the evaluation of expr to fail.

```lisp
(define-symbolic id type)
(define-symbolic* id type)
(assert expr)
(verify expr)
(debug [type ...+] expr)
(solve expr)
(synthesize #:forall expr #:guarantee expr)
```

**Why do poly and fact output different values on the input -6?**

```lisp
(require rosette/query/debug rosette/lib/render)
(define (poly x)
  (+ (* x x x x) (* 6 x x x) (* 11 x x) (* 6 x)))
(define/debug (fact x)
  (* x (+ x 1) (+ x 2) (+ x 2)))
(define (same p f x)
  (assert (= (p x) (f x))))

> (render ; visualize the result
  (debug [integer?]
    (same poly fact -6)))
```

To use `debug`, require the debugging libraries, mark fact as the candidate for debugging, save the module to a file, and issue a `debug` query.
Rosette constructs: debug

(debug) searches for a minimal set of expressions of the given types that cause the evaluation of expr to fail.

(define-symbolic id type)
(define-symbolic* id type)
(assert expr)
(verify expr)
(debug [type ...+] expr)
(solve expr)
(synthesize #:forall expr #:guarantee expr)

Why do poly and fact output different values on the input -6?

(require rosette/query/debug rosette/lib/render)
(define (poly x)
  (+ (* x x x x) (* 6 x x x)
     (* 11 x x) (* 6 x)))
(define/debug (fact x)
  (* x (+ x 1) (+ x 2) (+ x 2)))
(define (same p f x)
  (assert (= (p x) (f x))))

> (render ; visualize the result
  (debug [integer?]
    (same poly fact -6)))

To use debug, require the debugging libraries, mark fact as the candidate for debugging, save the module to a file, and issue a debug query.
Rosette constructs: from debug to solve

Can we repair fact on the input -6 as suggested by debug?

```scheme
(define (poly x)
  (+ (* x x x x) (* 6 x x x)
      (* 11 x x) (* 6 x)))

(define (fact x)
  (* x (+ x 1) (+ x 2) (+ x 2)))

(define (same p f x)
  (assert (= (p x) (f x))))
```
**Rosette constructs: from debug to solve**

`solve` searches for a binding of symbolic constants to concrete values that causes all assertions in `expr` to pass.

```
(define-symbolic id type)
(define-symbolic* id type)
(assert expr)
(verify expr)
(debug [type ...+] expr)
(solve expr)
(synthesize
 #:forall expr
 #:guarantee expr)
```

Can we repair `fact` on the input -6 as suggested by `debug`?

```
(define (poly x)
  (+ (* x x x x) (* 6 x x x)
      (* 11 x x) (* 6 x)))
(define (fact x)
  (* x (+ x 1) (+ x 2) (+ x 2)))
(define (same p f x)
  (assert (= (p x) (f x))))
```
**Rosette constructs: solve**

`solve` searches for a binding of symbolic constants to concrete values that causes all assertions in `expr` to pass.

```
(define-symbolic id type)
(define-symbolic* id type)
(assert expr)
(verify expr)
(debug [type ...+] expr)
(solve expr)
(synthesize #:forall expr #:guarantee expr)
```

Can we repair `fact` on the input -6 as suggested by `debug`?

```
(define (poly x)
  (+ (* x x x x) (* 6 x x x)
      (* 11 x x) (* 6 x)))

(define (fact x)
  (define-symbolic* c1 c2 c3 integer?)
  (* (+ x c1) (+ x 1) (+ x c2) (+ x c3)))

(define (same p f x)
  (assert (= (p x) (f x))))

> (solve (same poly fact -6))
```
Rosette constructs: solve

`solve` searches for a binding of symbolic constants to concrete values that causes all assertions in `expr` to pass.

```scheme
(define-symbolic id type)
(define-symbolic* id type)
(assert expr)
(verify expr)
(debug [type ...+] expr)
(solve expr)
(synthesize #:forall expr #:guarantee expr)
```

Can we repair fact on the input -6 as suggested by `debug`?

```scheme
(define (poly x)
  (+ (* x x x x) (* 6 x x x)
      (* 11 x x) (* 6 x)))

(define (fact x)
  (define-symbolic* c1 c2 c3 integer?)
  (* (+ x c1) (+ x 1) (+ x c2) (+ x c3)))

(define (same p f x)
  (assert (= (p x) (f x))))

> (solve (same poly fact -6))
(model [c1$0 -66] [c2$0 7] [c3$0 7])
```

Yes! The solver finds concrete values for `c1`, `c2`, and `c3` that work for the input -6.
Rosette constructs: solve many with define-symbolic*

**solve** searches for a binding of symbolic constants to concrete values that causes all assertions in **expr** to pass.

```scheme
(define-symbolic id type)
(define-symbolic* id type)
```

**assert** **expr**

**verify** **expr**

**debug** [type ...+] **expr**

**solve** **expr**

**synthesize**

```scheme
#:forall expr
#:guarantee expr
```

Can we repair fact on multiple inputs individually?

```scheme
(define (poly x)
  (+ (* x x x x) (* 6 x x x)
     (* 11 x x) (* 6 x)))

(define (fact x)
  (define-symbolic* c1 c2 c3 integer?)
  (* (+ x c1) (+ x 1) (+ x c2) (+ x c3)))

(define (same p f x)
  (assert (= (p x) (f x))))

> (solve (begin
    (same poly fact -6)
    (same poly fact 12)))

(model [c1$1 -66] [c2$1 7] [c3$1 7]
      [c1$2 2508] [c2$2 -11] [c3$2 -11])
```

Solving same for multiple inputs: note the behavior of define-symbolic*.
Rosette constructs: solve many with define-symbolic

- `solve expr` searches for a binding of symbolic constants to concrete values that causes all assertions in `expr` to pass.

```
(define-symbolic id type)
(define-symbolic* id type)
(assert expr)
(verify expr)
(debug [type ...+] expr)
(solve expr)
(synthesize
 #:forall expr
 #:guarantee expr)
```

Can we repair fact on multiple inputs simultaneously?

```
(define (poly x)
  (+ (* x x x x) (* 6 x x x)
    (* 11 x x) (* 6 x)))

(define (fact x)
  (define-symbolic c1 c2 c3 integer?)
  (* (+ x c1) (+ x 1) (+ x c2) (+ x c3)))

(define (same p f x)
  (assert (= (p x) (f x))))

> (solve (begin
           (same poly fact -6)
           (same poly fact 12))
       (model [c1 2] [c2 3] [c3 0]))
```

Solving same for multiple inputs: note the behavior of `define-symbolic`.

Rosette constructs: from solve to synthesize

Can we repair fact on all inputs as suggested by solve?

```
(define (poly x)
  (+ (* x x x x) (* 6 x x x)
      (* 11 x x) (* 6 x)))

(define (fact x)
  (define-symbolic c1 c2 c3 integer?)
  (+ (+ x c1) (+ x 1) (+ x c2) (+ x c3)))

(define (same p f x)
  (assert (= (p x) (f x))))
```
Rosette constructs: synthesize

*synthesize* searches for a binding that causes all assertions in #:guarantee expr to pass for all bindings of the symbolic constants in the #:forall expr.

```
(define-symbolic id type)
(define-symbolic* id type)
(assert expr)
(verify expr)
(debug [type ...+] expr)
(solve expr)
(synthesize #:forall expr #:guarantee expr)
```

Can we repair fact on all inputs as suggested by *solve*?

```
(define (poly x)
  (+ (* x x x x) (* 6 x x x)
      (* 11 x x) (* 6 x)))

(define (fact x)
  (define-symbolic c1 c2 c3 integer?)
  (* (+ x c1) (+ x 1) (+ x c2) (+ x c3)))

(define (same p f x)
  (assert (= (p x) (f x))))

> (define-symbolic* i integer?)
> (synthesize
  #:forall i
  #:guarantee (same poly fact i))
```
Rosette constructs: synthesize

synthesize searches for a binding that causes all assertions in #:guarantee expr to pass for all bindings of the symbolic constants in the #:forall expr.

(define-symbolic id type)
(define-symbolic* id type)
(assert expr)
(verify expr)
(debug [type ...+] expr)
(solve expr)
(synthesize
  #:forall expr
  #:guarantee expr)

Yes! The solver finds concrete values for c1, c2, and c3 that work for every input i.

Can we repair fact on all inputs as suggested by solve?

(define (poly x)
  (+ (* x x x x) (* 6 x x x)
      (* 11 x x) (* 6 x)))

(define (fact x)
  (define-symbolic c1 c2 c3 integer?)
  (* (+ x c1) (+ x 1) (+ x c2) (+ x c3)))

(define (same p f x)
  (assert (= (p x) (f x))))

> (define-symbolic* i integer?)
> (synthesize
  #:forall i
  #:guarantee (same poly fact i))
(model [c1 3] [c2 0] [c3 2])
Rosette constructs: synthesize

synthesize searches for a binding that causes all assertions in #:guarantee expr to pass for all bindings of the symbolic constants in the #:forall expr.

(define-symbolic id type)
(define-symbolic* id type)
(assert expr)
(verify expr)
(debug [type ...+] expr)
(solve expr)
(synthesize #:forall expr #:guarantee expr)

Can we repair fact on all inputs as suggested by solve?

(require rosette/lib/synthax)
(define (poly x)
  (+ (* x x x x) (* 6 x x x) (* 11 x x) (* 6 x)))
(define (fact x)
  (* (+ x (??)) (+ x 1) (+ x (??)) (+ x (??))))
(define (same p f x)
  (assert (= (p x) (f x))))

> (define-symbolic* i integer?)
> (print-forms ; print the generated code
  (synthesize #:forall i #:guarantee (same poly fact i)))

To generate code, require the sketching library, save the module to a file, and issue a synthesize query.
Rosette constructs: synthesize

synthesize searches for a binding that causes all assertions in #:guarantee expr to pass for all bindings of the symbolic constants in the #:forall expr.

(define-symbolic id type)
(define-symbolic* id type)
(assert expr)
(verify expr)
(debug [type ...+] expr)
(solve expr)
(synthesize #:forall expr #:guarantee expr)

Can we repair fact on all inputs as suggested by solve?

(require rosette/lib/synthax)
(define (poly x)
  (+ (* x x x x) (* 6 x x x) (* 11 x x) (* 6 x))
)
(define (fact x)
  (* (+ x 3) (+ x 1) (+ x 0) (+ x 2)))
(define (same p f x)
  (assert (= (p x) (f x)))))

> (define-symbolic* i integer?)
> (print-forms ; print the generated code
  (synthesize #:forall i #:guarantee (same poly fact i)))
A programming model that integrates solvers into the language, providing constructs for program verification, synthesis, and more.

**Solver-aided programming in two parts:**
1. **getting started**
2. **going pro**

How to use a solver-aided language: the workflow, constructs, and **gotchas**.

How to build your own solver-aided tool via direct symbolic evaluation or language embedding.
Common pitfalls and gotchas

“A gotcha is a valid construct in a system, program or programming language that works as documented but is counter-intuitive and almost invites mistakes because it is both easy to invoke and unexpected or unreasonable in its outcome.”

—Wikipedia
Common pitfalls and gotchas: reasoning precision

Reasoning precision

Unbounded loops

Unsafe features

- Determines if integers and reals are approximated using k-bit words or treated as infinite-precision values.
- Controlled by setting current-bitwidth to an integer \( k > 0 \) or \#f for approximate or precise reasoning, respectively.
Common pitfalls and gotchas: reasoning precision

Reasoning precision
Unbounded loops
Unsafe features

• Determines if integers and reals are approximated using k-bit words or treated as infinite-precision values.
• Controlled by setting `current-bitwidth` to an integer `k > 0` or `#f` for approximate or precise reasoning, respectively.

; default current-bitwidth is #f
(define-symbolic x integer?)
(solve (assert (= x 64)))
Common pitfalls and gotchas: reasoning precision

Reasoning precision

Unbounded loops

Unsafe features

- Determines if integers and reals are approximated using k-bit words or treated as infinite-precision values.
- Controlled by setting current-bitwidth to an integer \( k > 0 \) or \(#f\) for approximate or precise reasoning, respectively.

```scheme
;; default current-bitwidth is #f
> (define-symbolic x integer?)
> (solve (assert (= x 64)))
(model [x 64])
```
Common pitfalls and gotchas: reasoning precision

Reasoning precision

Unbounded loops

Unsafe features

• Determines if integers and reals are approximated using k-bit words or treated as infinite-precision values.

• Controlled by setting current-bitwidth to an integer k > 0 or #f for approximate or precise reasoning, respectively.

; default current-bitwidth is #f
> (define-symbolic x integer?)
> (solve (assert (= x 64)))
(model [x 64])
> (verify (assert (not (= x 64))))
Common pitfalls and gotchas: reasoning precision

Reasoning precision

Unbounded loops

Unsafe features

• Determines if integers and reals are approximated using k-bit words or treated as infinite-precision values.

• Controlled by setting current-bitwidth to an integer \( k > 0 \) or \#f for approximate or precise reasoning, respectively.

; default current-bitwidth is \#f
> (define-symbolic x integer?)
> (solve (assert (= x 64)))
(model [x 64])
> (verify (assert (not (= x 64))))
(model [x 64])
Common pitfalls and gotchas: reasoning precision

Reasoning precision

Unbounded loops

Unsafe features

- Determines if integers and reals are approximated using k-bit words or treated as infinite-precision values.
- Controlled by setting current-bitwidth to an integer $k > 0$ or #f for approximate or precise reasoning, respectively.

; default current-bitwidth is #f
> (define-symbolic x integer?)
> (solve (assert (= x 64)))
(model [x 64])
> (verify (assert (not (= x 64))))
(model [x 64])
> (current-bitwidth 5)
> (solve (assert (= x 64)))
Common pitfalls and gotchas: reasoning precision

Reasoning precision

Unbounded loops

Unsafe features

• Determines if integers and reals are approximated using k-bit words or treated as infinite-precision values.

• Controlled by setting current-bitwidth to an integer \( k > 0 \) or \(#f\) for approximate or precise reasoning, respectively.

```lisp
; default current-bitwidth is #f
> (define-symbolic x integer?)
> (solve (assert (= x 64)))
(model [x 64])
> (verify (assert (not (= x 64))))
(model [x 64])

> (current-bitwidth 5)
> (solve (assert (= x 64)))
(model [x 0])
> (verify (assert (not (= x 64))))
(model [x 0])
```
Common pitfalls and gotchas: unbounded loops

Reasoning precision

Unbounded loops

Unsafe features

- Loops and recursion must be bounded (aka self-finitizing) by
  - concrete termination conditions, or
  - upper bounds on size of iterated (symbolic) data structures.

- Unbounded loops and recursion run forever.
Common pitfalls and gotchas: unbounded loops

Reasoning precision

Unbounded loops

Unsafe features

- Loops and recursion must be bounded (aka self-finitizing) by
  - concrete termination conditions, or
  - upper bounds on size of iterated (symbolic) data structures.
- Unbounded loops and recursion run forever.

(define (search x xs)
  (cond
    [(null? xs) #f]
    [(equal? x (car xs)) #t]
    [else (search x (cdr xs))])))

> (define-symbolic xs integer? [5])
> (define-symbolic xl i integer?)
> (define ys (take xs xl))
> (verify
  (when (<= 0 i (- xl 1))
    (assert (search (list-ref ys i) ys)))))
Common pitfalls and gotchas: unbounded loops

Reasoning precision

Unbounded loops

Unsafe features

- Loops and recursion must be bounded (aka self-finitizing) by
  - concrete termination conditions, or
  - upper bounds on size of iterated (symbolic) data structures.
- Unbounded loops and recursion run forever.

```scheme
(define (search x xs)
  (cond
    [(null? xs) #f]
    [(equal? x (car xs)) #t]
    [else (search x (cdr xs))]))

> (define-symbolic xs integer? [5])
> (define-symbolic xl i integer?)
> (define ys (take xs xl))
> (verify
  (when (<= 0 i (- xl 1))
    (assert (search (list-ref ys i) ys))))
(unsat)

Terminates because search iterates over a bounded structure.
Common pitfalls and gotchas: unbounded loops

Reasoning precision

Unbounded loops

Unsafe features

- Loops and recursion must be *bounded* (aka *self-finitizing*) by
  - concrete termination conditions, or
  - upper bounds on size of iterated (symbolic) data structures.
- Unbounded loops and recursion run forever.

```
(define (factorial n)
  (cond
    [ (= n 0) 1]
    [else (* n (factorial (- n 1)))]))
```
Common pitfalls and gotchas: unbounded loops

Reasoning precision

Unbounded loops

Unsafe features

- Loops and recursion must be **bounded** (aka self-finitizing) by
  - concrete termination conditions, or
  - upper bounds on size of iterated (symbolic) data structures.
- Unbounded loops and recursion run forever.

(define (factorial n)
  (cond
    [(= n 0) 1]
    [else (* n (factorial (- n 1)))])

> (define-symbolic k integer?)
> (solve
  (assert (> (factorial k) 10)))
Common pitfalls and gotchas: unbounded loops

Reasoning precision

Unbounded loops

Unsafe features

- Loops and recursion must be **bounded** (aka self-finitizing) by
  - concrete termination conditions, or
  - upper bounds on size of iterated (symbolic) data structures.
- Unbounded loops and recursion run forever.

```
(define (factorial n)
  (cond
    [(= n 0) 1]
    [else (* n (factorial (- n 1)))]))
```

```scheme
(define-symbolic k integer?)
(solve
  (assert (> (factorial k) 10)))
```

Unbounded because factorial termination depends on k.
Common pitfalls and gotchas: unbounded loops

Reasoning precision

Unbounded loops

Unsafe features

- Loops and recursion must be *bounded* (aka *self-finitizing*) by
  - concrete termination conditions, or
  - upper bounds on size of iterated (symbolic) data structures.
- Unbounded loops and recursion run forever.

Bound the recursion with a concrete guard.

```lisp
(define (factorial n g)
  (assert (>= g 0))
  (cond
    [(= n 0) 1]
    [else (* n (factorial (- n 1) (- g 1)))]))
```
Common pitfalls and gotchas: unbounded loops

Reasoning precision

Unbounded loops

Unsafe features

- Loops and recursion must be bounded (aka self-finitizing) by
  - concrete termination conditions, or
  - upper bounds on size of iterated (symbolic) data structures.
- Unbounded loops and recursion run forever.

Bound the recursion with a concrete guard.

```
(define (factorial n g)
  (assert (>= g 0))
  (cond
    [(= n 0) 1]
    [else (* n (factorial (- n 1) (- g 1)))]))
```

```
> (define-symbolic k integer?)
> (solve
  (assert (> (factorial k 3) 10)))
```
Common pitfalls and gotchas: unbounded loops

Reasoning precision

Unbounded loops

Unsafe features

- Loops and recursion must be bounded (aka self-finitizing) by
  - concrete termination conditions, or
  - upper bounds on size of iterated (symbolic) data structures.
- Unbounded loops and recursion run forever.

Bound the recursion with a concrete guard.

(define (factorial n g)
  (assert (>= g 0))
  (cond
    [(= n 0) 1]
    [else (* n (factorial (- n 1) (- g 1)))]))

> (define-symbolic k integer?)
> (solve
  (assert (> (factorial k 3) 10)))
(unsat)

UNSAT because the bound is too small to find a solution.
Common pitfalls and gotchas: unbounded loops

Reasoning precision
Unbounded loops
Unsafe features

• Loops and recursion must be **bounded** (aka self-finitizing) by
  • concrete termination conditions, or
  • upper bounds on size of iterated (symbolic) data structures.
• Unbounded loops and recursion run forever.

Bound the recursion with a concrete guard.

```
(define (factorial n g)
  (assert (>= g 0))
  (cond
    [(= n 0) 1]
    [else (* n (factorial (- n 1) (- g 1)))]))
```

> (define-symbolic k integer?)
> (solve
  (assert (> (factorial k 4) 10)))

(model
  [k 4])

Make sure the bound is large enough …
Common pitfalls and gotchas: unsafe features

Reasoning precision
Unbounded loops
Unsafe features

- Rosette lifts only a core subset of Racket to operate on symbolic values. This includes all constructs in #lang rosette/safe

- Unlifted constructs can be used in #lang rosette but require care: the programmer must determine when it is okay for symbolic values to flow to unlifted code.
Common pitfalls and gotchas: unsafe features

Reasoning precision

Unbounded loops

Unsafe features

• Rosette lifts only a core subset of Racket to operate on symbolic values. This includes all constructs in #lang rosette/safe

• Unlifted constructs can be used in #lang rosette but require care: the programmer must determine when it is okay for symbolic values to flow to unlifted code.

; vectors are lifted
> (define v (vector 1 2))
> (define-symbolic k integer?)
> (vector-ref v k)
Common pitfalls and gotchas: unsafe features

Reasoning precision
Unbounded loops
Unsafe features

- Rosette lifts only a core subset of Racket to operate on symbolic values. This includes all constructs in #lang rosette/safe
- Unlifted constructs can be used in #lang rosette but require care: the programmer must determine when it is okay for symbolic values to flow to unlifted code.

; vectors are lifted
> (define v (vector 1 2))
> (define-symbolic k integer?)
> (vector-ref v k)
(ite* (¬ (= 0 k) 1) (¬ (= 1 k) 2)))
Common pitfalls and gotchas: unsafe features

Reasoning precision
Unbounded loops
Unsafe features

• Rosette *lifts* only a core subset of Racket to operate on symbolic values. This includes all constructs in `#lang rosette/safe`

• Unlifted constructs can be used in `#lang rosette` but require care: the programmer must determine when it is okay for symbolic values to flow to unlifted code.

```scheme
; vectors are lifted
> (define v (vector 1 2))
> (define-symbolic k integer?)
> (vector-ref v k)
(ite* (¬ (= 0 k)) 1 (¬ (= 1 k)) 2))

; hashes are unlifted
> (define h (make-hash '(((0 . 1)(1 . 2)))))
> (hash-ref h k)
```
Common pitfalls and gotchas: unsafe features

Reasoning precision

Unbounded loops

Unsafe features

• Rosette lifts only a core subset of Racket to operate on symbolic values. This includes all constructs in `#lang rosette/safe`

• Unlifted constructs can be used in `#lang rosette` but require care: the programmer must determine when it is okay for symbolic values to flow to unlifted code.

; vectors are lifted
> (define v (vector 1 2))
> (define-symbolic k integer?)
> (vector-ref v k)
(ite* (\(\neg\) (= 0 k) 1) (\(\neg\) (= 1 k) 2))

; hashes are unlifted
> (define h (make-hash '((0 . 1)(1 . 2))))
> (hash-ref h k)
hash-ref: no value found for key key: k
Common pitfalls and gotchas: unsafe features

Reasoning precision

Unbounded loops

Unsafe features

• Rosette lifts only a core subset of Racket to operate on symbolic values. This includes all constructs in \texttt{#lang rosette/safe}

• Unlifted constructs can be used in \texttt{#lang rosette} but require care: the programmer must determine when it is okay for symbolic values to flow to unlifted code.

\begin{verbatim}
; vectors are lifted
> (define v (vector 1 2))
> (define-symbolic k integer?)
> (vector-ref v k)
(ite* (\top (\equiv 0 k) 1) (\top (\equiv 1 k) 2))
\end{verbatim}

\begin{verbatim}
; hashes are unlifted
> (define h (make-hash '((0 . 1)(1 . 2))))
> (hash-ref h k)
hash-ref: no value found for key
key: k
> (hash-set! h k 3)
> (hash-ref h k)
\end{verbatim}
Common pitfalls and gotchas: unsafe features

Reasoning precision

Unbounded loops

Unsafe features

- Rosette *lifts* only a core subset of Racket to operate on symbolic values. This includes all constructs in `#lang rosette/safe`

- Unlifted constructs can be used in `#lang rosette` but require care: the programmer must determine when it is okay for symbolic values to flow to unlifted code.

; vectors are lifted
> (define v (vector 1 2))
> (define-symmetric k integer?)
> (vector-ref v k)
(ite* (¬ (= 0 k) 1) (¬ (= 1 k) 2))

; hashes are unlifted
> (define h (make-hash '(((0 . 1)(1 . 2))))
> (hash-ref h k)
**hash-ref: no value found for key**
  **key: k**
> (hash-set! h k 3)
> (hash-ref h k)
3
A programming model that integrates solvers into the language, providing constructs for program verification, synthesis, and more.

Solver-aided programming in two parts: (1) **getting started** and (2) going pro

How to use a solver-aided language: the workflow, constructs, and gotchas.

How to build your own solver-aided tool via direct symbolic evaluation or language embedding.
Summary

Today

• Course overview & logistics
• Getting started with solver-aided programming

Next lecture

• Going pro with solver-aided programming